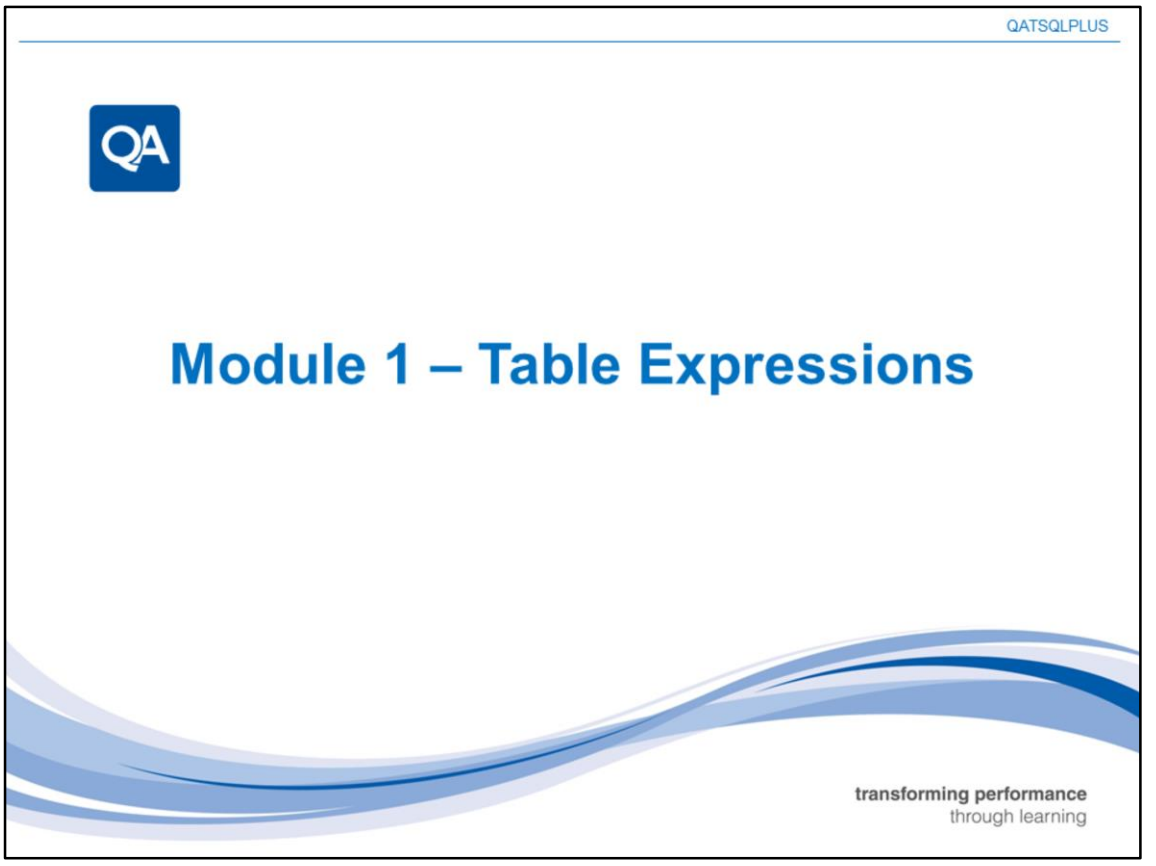
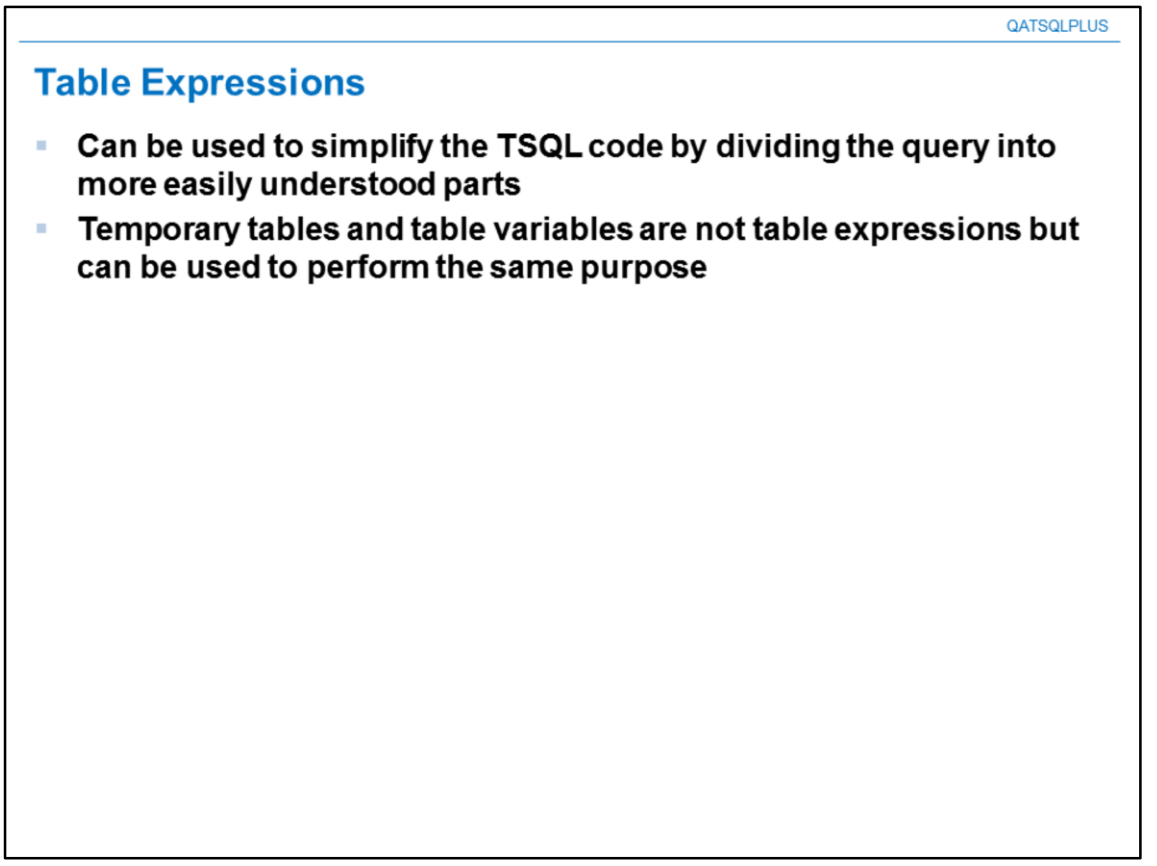

QATSQLPLUS
Module 1 – Table Expressions
transforming performance
through learning

Table Expressions

- Views
- Table-valued functions
- Derived tables
- Common table expressions
- Temporary tables
- Table variables
- Comparison



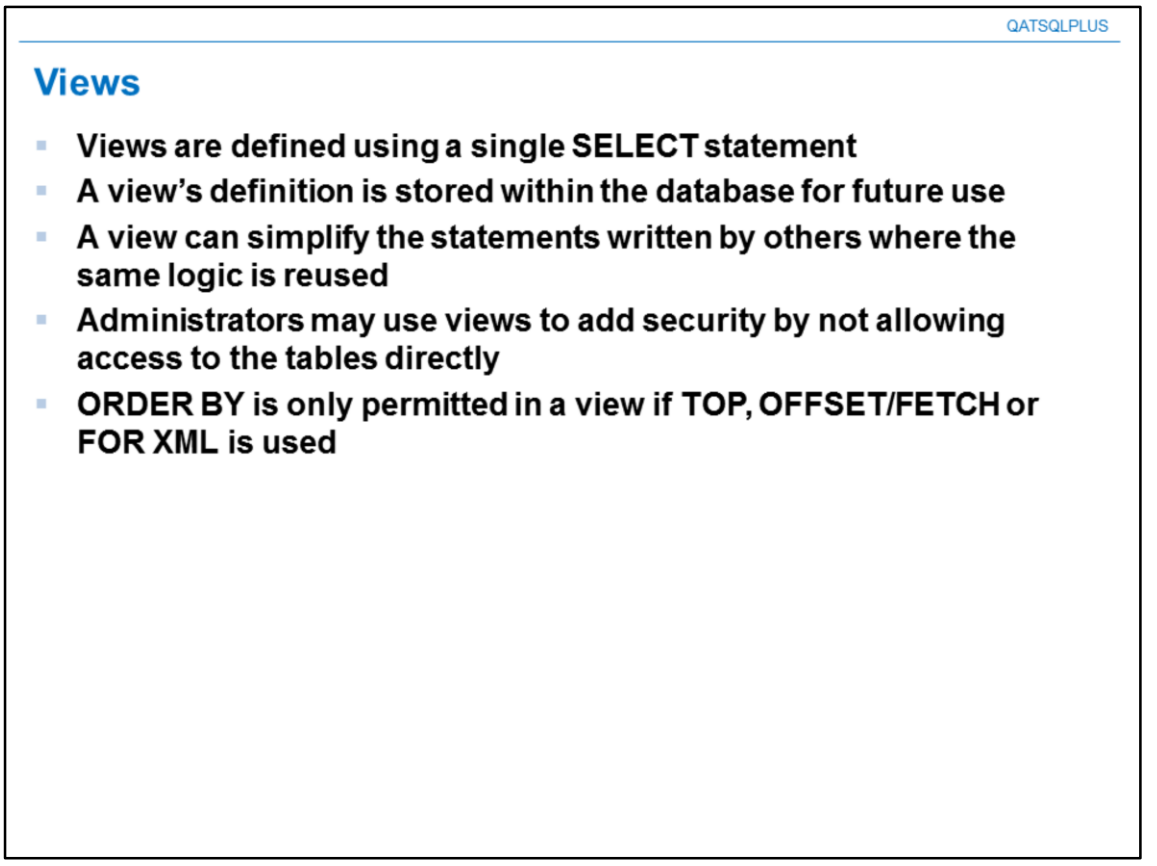
TSQL allow for multiple table expression types.

Table expressions can be used to divide the complicated query into more easily understood parts. This means that if you come back to query in the future you will be able to understand each part of the query or be able to update and change it easily.

The types of table expression covered in this module are:

- Views
- Table-valued functions
- Derived tables
- Common table expressions
- Temporary tables
- Table variables

Each of the table expressions listed above has advantages and disadvantages within TSQL. At the end of the module, a comparison table will be shown.



A view is a select query which has been pre-created and stored within the database. The view will not give up any performance advantages over the original query. You can simplify the use of a difficult query for other users where they do not have to provide the full query again, but can refer to the view by name. (Administrators may use view to provide a level of security above the underlying tables.)

As the views are called within select queries, by default, views cannot be ordered by clause. This is to avoid an expensive sort operation. An exception to this would be in conjunction with TOP, OFFSET or FOR XML queries as the order within a view dictates the rows returned within the view. For instance, if TOP was used without an order by the view, it may not return the correct rows.

Query below would return 50 rows. As no ORDER BY clause is included the query, it may return any 50 rows from the table.

```
CREATE VIEW dbo.Top50Rows AS
SELECT TOP 50 *
   FROM dbo.Products
```

Query below would return 50 rows with the highest ListPrice.

```
CREATE VIEW dbo.Top50Rows AS
SELECT TOP 50 *
   FROM dbo.Products
   ORDER BY ListPrice DESC
```

Creating Views

Command outline:

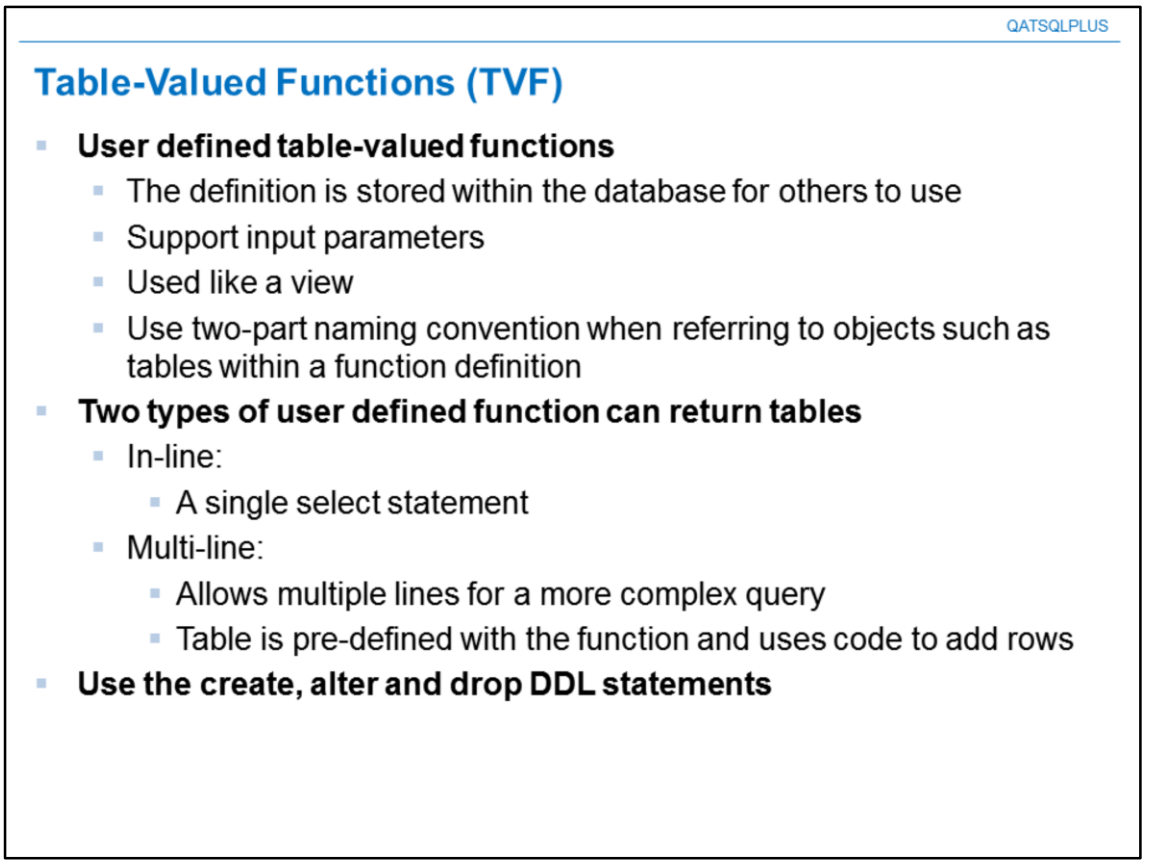
```
CREATE VIEW <viewname> AS
    SELECT <columns>
    FROM <table(s) including joins>
```

Demonstration:

```
CREATE VIEW dbo.SaleableProducts AS
    SELECT PSC.name AS Subcategory, ProductName, ListPrice, Color, Weight
    FROM Production.Product AS P
        INNER JOIN Production.ProductSubcategory AS PSC
            ON P.ProductSubcategoryID = PSC.ProductSubcategory
```

Use

```
SELECT * FROM dbo.SaleableProducts
```



User Defined Table-Valued Functions

- The definition is stored within the database for others to use
- Similar to a view that can accept parameters and return a dataset
- Support input parameters of types registered within the database – this could be in-built data types (int, varchar, datetime, etc.) or user-defined data types
    - The user-defined data types are beyond the scope of this course – more details on user-defined data types can be found here: https://msdn.microsoft.com/en-us/library/ms175007.aspx
- Use two-part naming convention when referring to objects such as tables within a function definition – as it is possible to include multiple tables with the same name (but different schemas), a two-part name should be used in both views and functions when referring to table names
    - A two-part name is recorded as SCHEMA.TABLENAME, such as HumanResources.Employee, where the schema is *HumanResources* and the table name is *Employee*

Two types of user defined function can return tables:

- In-line, where a single select query is executed based on the parameters
- Multi-line, (a more complex query) where the dataset is created using multiple statements and the structure of the table is pre-defined in the function header

Creating In-line TVF

Command outline:

```
CREATE FUNCTION <functionname> (<parameters>)
RETURNS TABLE () AS
    (SELECT <columns>
    FROM <table(s) including joins>)
```

Demonstration:

```
CREATE FUNCTION dbo.GetProductByColour(@Colour varchar(20))
    RETURNS TABLE AS
    RETURN(
        SELECT PSC.Name AS Subcategory, P.Name
            FROM Production.Product AS P
                INNER JOIN Production.ProductSubcategory AS PSC
                    ON P.ProductSubcategoryID = PSC.ProductSubcategoryID
            WHERE Color = @Colour
    )
```

Use:

```
SELECT * FROM dbo.GetProductsByColour('Red')
```

Creating Multi-Line TVF

Command outline:

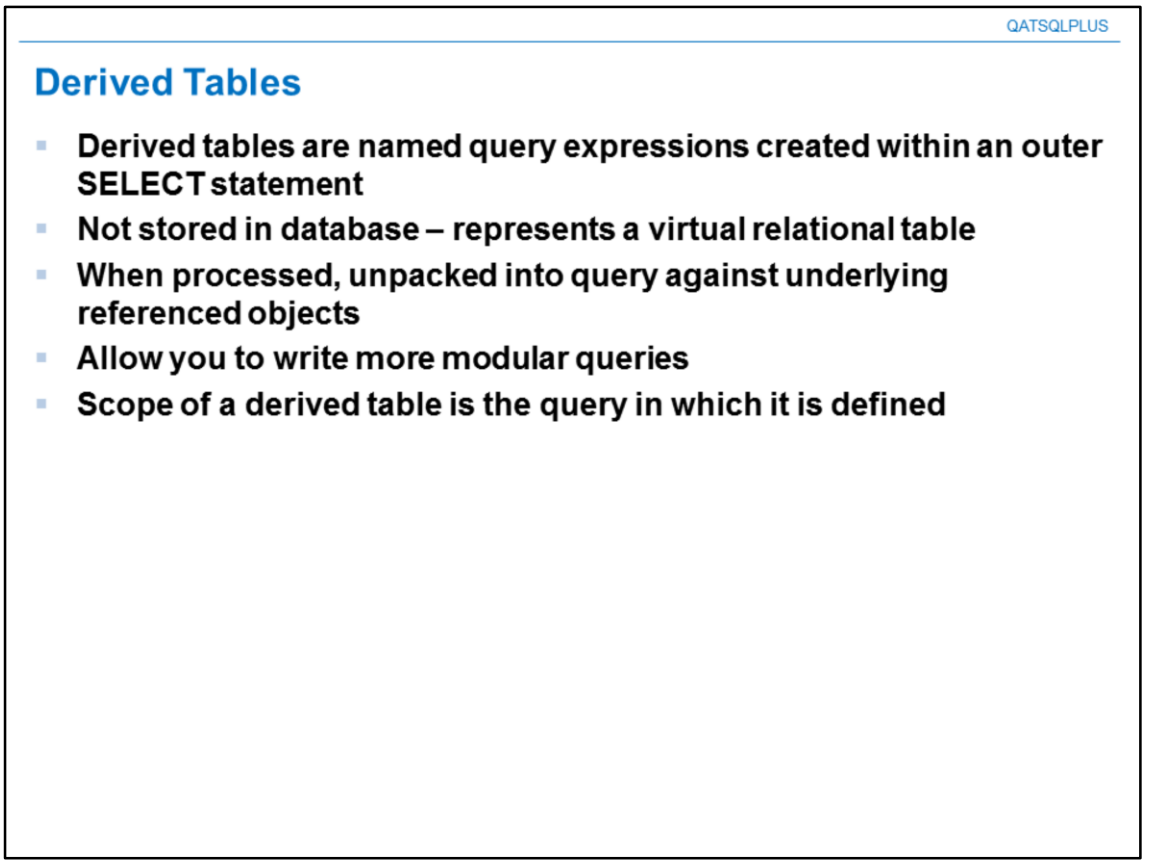
```
CREATE FUNCTION <functionname> (<parameters>)
RETURNS TABLE () AS
    (SELECT <columns>
    FROM <table(s) including joins>)
```

Demonstration:

```
ALTER FUNCTION dbo.GetProductByColours2(@Colour varchar(20))
    RETURNS @A TABLE(Subcategory varchar(100),Name varchar(100))
    AS
    BEGIN
        IF @Colour LIKE '%,%'
            RETURN
        INSERT INTO @A
            SELECT PSC.Name AS Subcategory, P.Name
                FROM Production.Product AS P
                    INNER JOIN Production.ProductSubcategory AS PSC
                        ON P.ProductSubcategoryID =
PSC.ProductSubcategoryID
                WHERE Color = @Colour
        RETURN
    END
```

Use:

```
SELECT * FROM dbo.GetProductsByColour2('Blue') -- returns all blue products
SELECT * FROM dbo.GetProductsByColour2('Blue,Black') -- returns empty set
```



Derived Tables

- Derived tables are named queries created within an outer SELECT statement

```
SELECT *
   FROM (
      SELECT <columns>
      FROM <table>
      ….. ) AS DerivedTableAlias
```

- Derived tables are processed with the outer query and are not stored in the database
- When processed, and unpacked into query against underlying referenced objects, the optimizer will calculate the most efficient way to execute the complete query
- Allows you to write more modular queries, where the derived table (inner query) may be executed separately from the outer query
    - An exception to this would be a correlated query where columns are passed from the outer query into the inner query
- Scope of a derived table is the query in which it is defined and cannot be referred to in any other query

Derived Table Rules

- **Must:**
  - Have an alias
  - Have unique names for all columns
  - Not use the order by clause unless in conjunction with top, offset / fetch or for XML
- **Column names**
  - Defined as part of the inner query

```
SELECT *
    FROM (
        SELECT ProductSubcategoryID, avg(ListPrice) AS AvgPrice
            FROM Production.Product
            GROUP BY ProductSubCategoryID
    ) AS Derivedtable
```

  - Defined as part of the alias definition

```
SELECT *
    FROM (
        SELECT ProductSubcategoryID, avg(ListPrice)
            FROM Production.Product
            GROUP BY ProductSubCategoryID
    ) AS Derivedtable (SubcategoryID, AvgPrice)
```

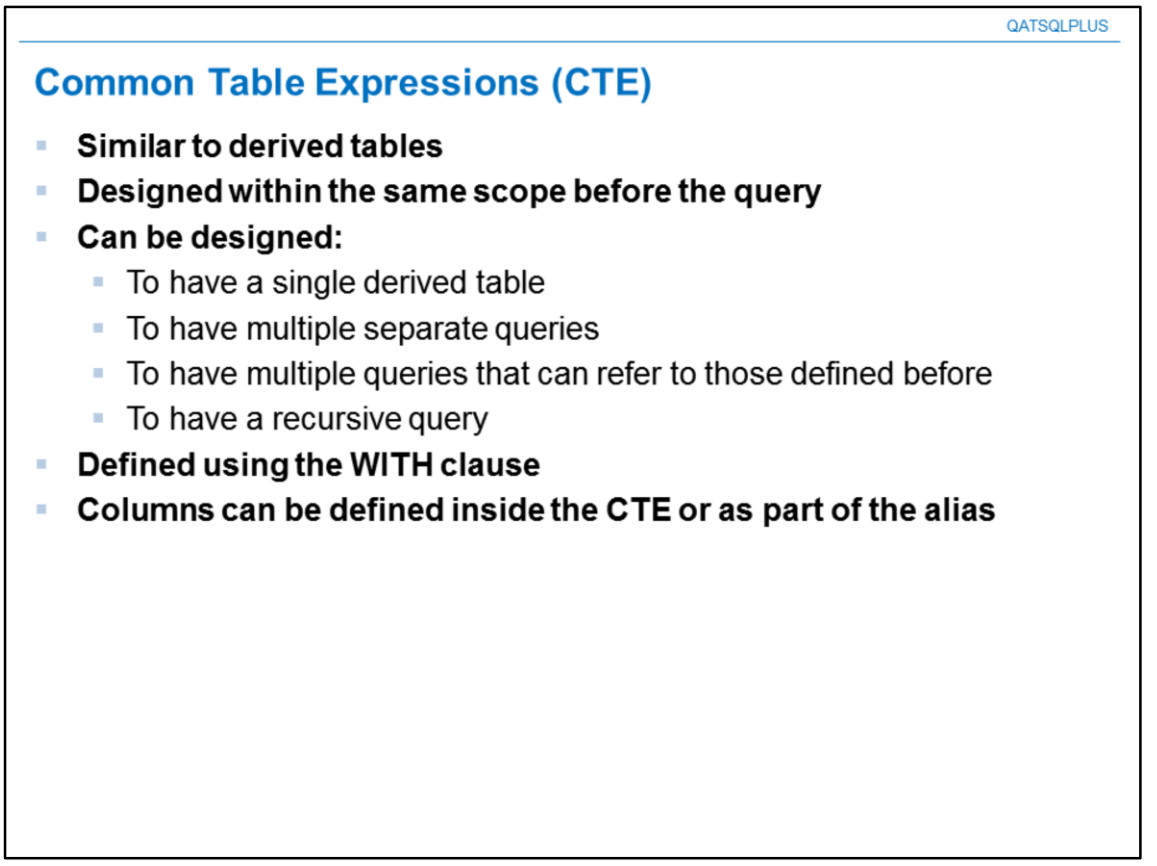
Derived Table Example

- **Show all products with above average list price for their subcategory**

```
SELECT P.ProductID, P.Name, P.ListPrice, DT.AvgPrice
    FROM Production.Product AS P
    INNER JOIN
    (
        SELECT ProductSubcategoryID, avg(ListPrice) AS AvgPrice
            FROM Production.Product
            GROUP BY ProductSubcategoryID
    ) AS DT
    ON P.ProductSubcategoryID = DT.ProductSubcategoryID
    WHERE P.ListPrice >= DT.AvgPrice
```

- **Derived table columns can be used within the SELECT clause**
- **Alternative which returns the same rows but does not allow the columns to be used within the SELECT clause**

```
SELECT P.ProductID, P.Name, P.ListPrice
    FROM  Production.Product AS P
    WHERE P.ListPrice >= (
        SELECT avg(ListPrice) AS AvgPrice
            FROM Production.Product AS P2
            WHERE P.ProductSubcategoryID = P2.ProductSubcategoryID
    )
```



Common Table Expressions (CTE)

- CTEs are similar to derived tables, but are defined once at the start of the query, and then referred to by name in later parts of the query
- Designed within the same scope before the query
- Defined using the WITH clause
- The CTE structure is flexible so that it can be used in multiple forms:
    - To have a single derived table
    - To have multiple separate queries
    - To have multiple queries that can refer to those defined before
    - To have a recursive query
- Column names can be defined inside the CTE or as part of the alias

CTE with column names declared

```
WITH CTEName1 (ColumnName1, ColumnName2, ….) AS
   ( <query> )
SELECT * FROM CTEName1;
```

CTE with column names inherited from inner query

```
WITH CTEName2 AS
   ( <query with column aliases> )
SELECT * FROM CTEName2
```

Common Table Expression (Basic)

Command outline:

```
WITH CTE_Name AS
(
    QueryDefinition
)
SELECT * FROM CTE_Name
```

Demonstration:

```
WITH Averages AS (
    SELECT ProductSubcategoryID, avg(ListPrice) AS AvgPrice
        FROM Production.Product
        GROUP BY ProductSubcategoryID
)
SELECT P.ProductID, P.Name, P.ListPrice, A.AvgPrice
    FROM Production.Product AS P
        INNER JOIN Averages AS A
            ON P.ProductSubcategoryID = DT.ProductSubcategoryID
    WHERE P.ListPrice >= A.AvgPrice
```

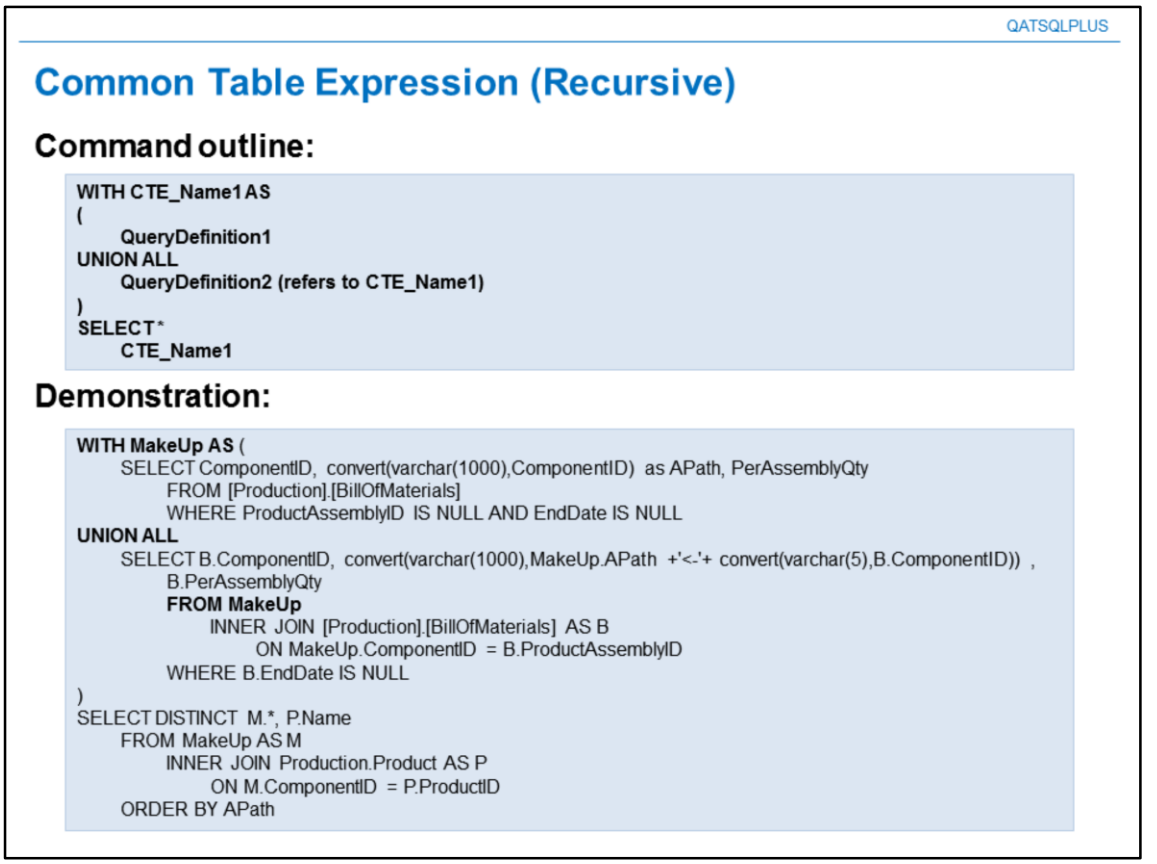
Common Table Expression (Multiple Level)

Command outline:

```
WITH CTE_Name1 AS
(
    QueryDefinition1
),
CTE_Name2 AS
(
    QueryDefinition2 (can refer to previous CTE)
)
SELECT *
    <<refer to any of the CTEs>>
```

Demonstration:

```
WITH
Top3Products AS (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY ProductSubcategoryID ORDER BY ListPrice
DESC) AS RN
        FROM Production.Product),
AveragePrice AS (
    SELECT ProductSubcategoryID, avg(ListPrice) AS AvgListPrice
        FROM Top3Products
        WHERE RN<=3
        GROUP BY ProductSubcategoryID
    )
SELECT *
    FROM AveragePrice
```



Demonstration Notes

The CTE is designed to produce a bill of materials list for the production of each product.

- Within the MakeUp CTE, the first query (before the UNION ALL) selects the products that are not components of other products and are currently saleable

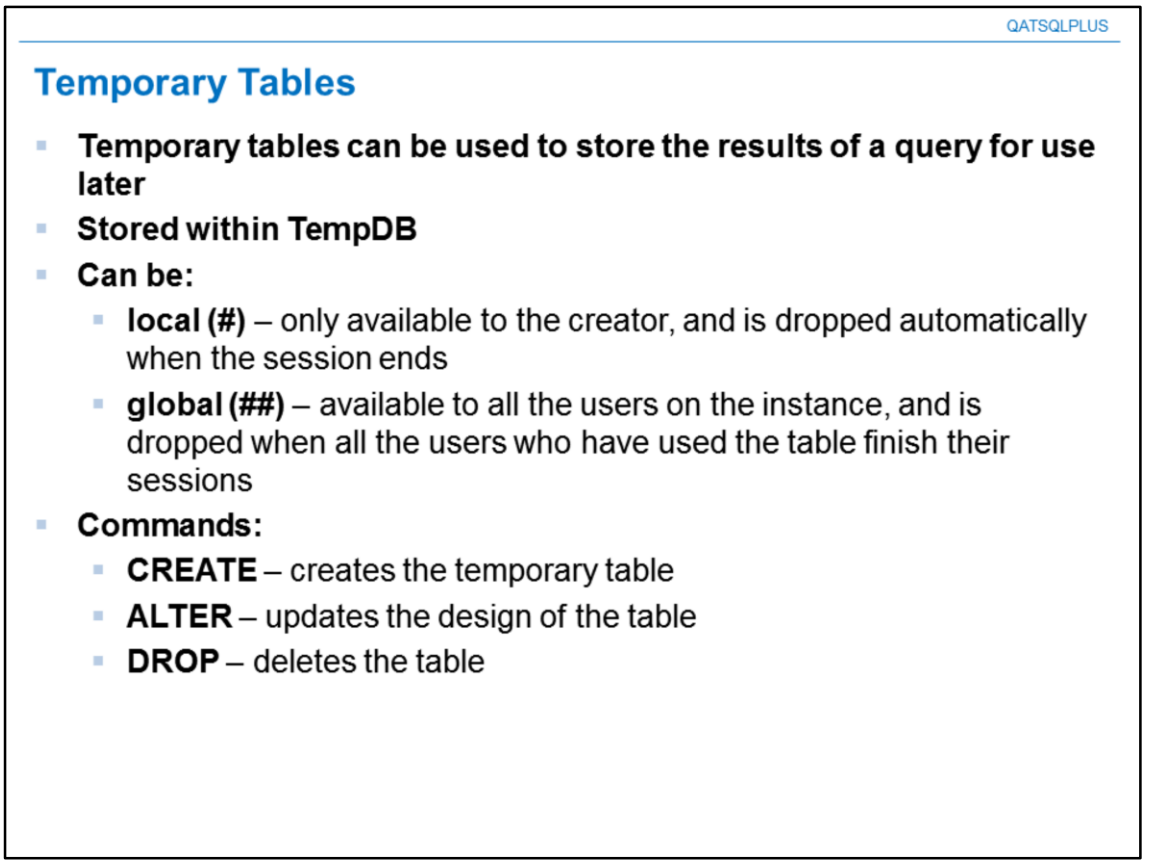
```
SELECT ComponentID, convert(varchar(1000),ComponentID) as APath, PerAssemblyQty
        FROM [Production].[BillOfMaterials]
        WHERE ProductAssemblyID IS NULL AND EndDate IS NULL
```

- Starting from the top level products each component level is found using a join to the current list

```
SELECT B.ComponentID, convert(varchar(1000),MakeUp.APath +'<-'+ convert(varchar(5),B.ComponentID)) ,
        B.PerAssemblyQty
        FROM MakeUp
            INNER JOIN [Production].[BillOfMaterials] AS B
                ON MakeUp.ComponentID = B.ProductAssemblyID
        WHERE B.EndDate IS NULL
```

- The final query returns the required products to the developer

```
SELECT DISTINCT M.*, P.Name
    FROM MakeUp AS M
        INNER JOIN Production.Product AS P
            ON M.ComponentID = P.ProductID
    ORDER BY APath
```



Temporary Tables

- Temporary tables can be used to store the results of a query for use later unlike the other table expressions
- Stored within TempDB for use later – this may cause performance issues if many temporary tables are created, used and removed by many users
- Can be:
    - **local (#)** – only available to the creator, and is dropped automatically when their session ends or is manually dropped
        - In the TempDB database, the name will have a uniqueidentifier added automatically so that multiple users can have local temporary tables with the same name
    - **global (##)** – available to all the users on the instance, and is dropped when all the users who have used the table finish their sessions
        - Each temporary table must be uniquely named
- Care should be taken when wanting to access temporary tables from stored procedures as these would execute in their own scope, and so would not be able to access local temporary tables
- Commands:
    - **CREATE** – creates the temporary table
    - **ALTER** – updates the design of the table
    - **DROP** – deletes the table

Temporary Tables

Command outline:

```
CREATE TABLE (# or ##)TableName(
      column definitions
)

INSERT INTO (# or ##)TableName
      SELECT / VALUES to insert

DROP TABLE (# or ##)TableName
```

Demonstration:

```
CREATE TABLE #Averages(
      SubcategoryID int,
      AverageListPrice money
)
GO

INSERT INTO #Averages
      SELECT ProductSubcategoryID, avg(ListPrice) AS AvgPrice
                  FROM Production.Product
                  GROUP BY ProductSubcategoryID
GO

SELECT *
      FROM #Averages
GO

DROP TABLE #Averages
```

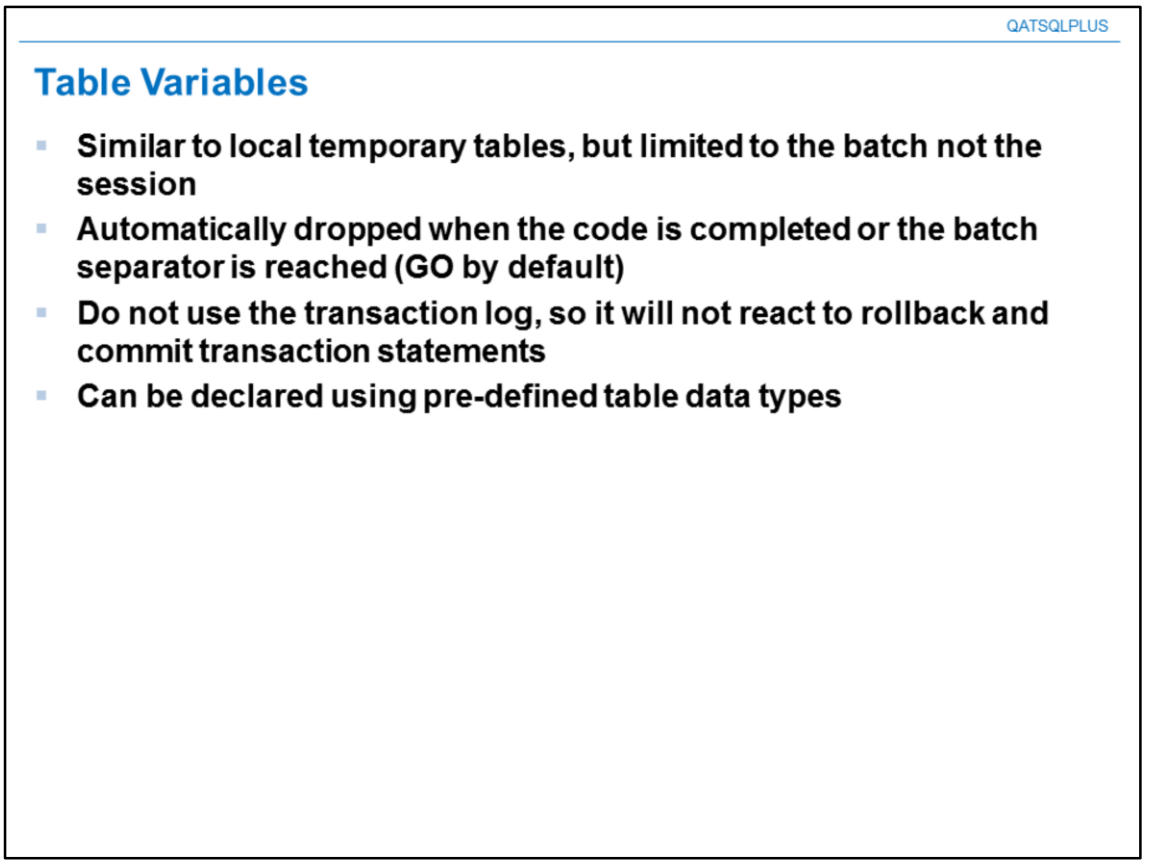


Table Variables

- Similar to local temporary tables, but limited to the batch not the session
- Automatically dropped when the code is completed or the batch separator is reached (GO by default)
- Do not use the transaction log, so it will not react to rollback and commit transaction statements
- Use the DECLARE statement to declare the table variable

```
DECLARE @TableVariableName TABLE (tabledefinition)
```

- Can be declared using pre-defined table data types which makes the table definition reusable by name

```
CREATE TYPE dbo.CustomTypeName AS TABLE (tabledefinition)

DECLARE @TableVariableName dbo.CustomTypeName
```

Table Variables

Command outline:

```
DECLARE @varName TABLE (
    column definitions
)

INSERT INTO @varName
    SELECT / VALUES to insert
```

Demonstration:

```
DECLARE @Averages(
    SubcategoryID int,
    AverageListPrice money
)

INSERT INTO @Averages
    SELECT ProductSubcategoryID, avg(ListPrice) AS AvgPrice
        FROM Production.Product
        GROUP BY ProductSubcategoryID

SELECT * FROM @Averages
GO

SELECT * FROM @Averages – an error occurs as the variable was dropped at the previous GO
statement
```

Comparison Table

	Views	Table-valued function	Derived tables	Common table expression	Temporary table	Table variable
Main use	Storing the definition of a query for use later	Storing the definition of code for use later	Writing more complex queries than are possible normally	Simplifying code by allowing the query to be deconstructed into smaller queries	Storing a dataset for reuse later, by the session owner or other sessions	Storing a dataset for reuse later in the same batch
Definition stored inside database	●	●				
Data exists	Single execution of the view	Single execution of the TVF	Single execution of the query	Single execution of the outer query	Session	Batch
Design shared with others	●	●			Depends on type	
Recursive				●		
Allow parameters to be passed		●				
Allows access to declared variables		Only declared variables within the function Declared variables cannot be used with in-line TVF	●	●		

Exercise